2012

RAMPS: reconfigurable architecture for minimal perfect sequencing using the Convey hybrid core computer

Chad Michael Nelson
Iowa State University

RAMPS: reconfigurable architecture for minimal perfect sequencing using the Convey hybrid core computer

by

Chad Michael Nelson

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph Zambreno, Major Professor
Phillip Jones
Heike Hofmann

Iowa State University
Ames, Iowa
2012

DEDICATION

For Erin, family, and friends... thank you. Life would not be so sweet without you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

Thank you to my committee for their support, efforts, and contributions to this work: Dr. Zambreno, Dr. Jones, and Dr. Hofmann. To Dr. Zambreno and Dr. Jones, for their patience, enduring support, and ability to bring out my competitive spirit. Special thanks to Kevin Townsend.

ABSTRACT

The alignment of many short sequences of DNA, called reads, to a long reference genome is a common task in molecular biology. When the problem is expanded to handle typical workloads of billions of reads, execution time becomes critical. While existing solutions attempt to align a high percentage of the reads using a small memory footprint, RAMPS (Reconfigurable Architecture for Minimal Perfect Sequencing) focuses on perform fast exact matching. Using the human genome as a reference, RAMPS aligns short reads on the order of hundreds of thousands of times faster than current software implementations such as SOAP2 or Bowtie, and about 1000 times faster than GPU implementations such as SOAP3. Whereas other aligners require hours to preprocess reference genomes, RAMPS can preprocess the human genome in a few minutes, opening doors via the ability to use arbitrary reference sources for alignment and increasing the amount of data that aligns with the reference.

CHAPTER 1. OVERVIEW

We are in a golden era of DNA research. After the first human genome was sequenced in 2003 [23], the next generation of sequencing technologies were developed with higher throughput and lower costs. The machines operate by breaking a person's genome into smaller fragments, or "reads". Since humans share 99.9% of their DNA, a reference genome is used to help align the reads in the correct order. Many reads, copies of the same genome, are aligned to improve the quality of the process. These techniques have produced a massive amount of data needing alignment.

For example, a single Illumina HiSeq machine sequences 120 billion base pairs in a 27 hours run [17]., and the Beijing Genomics Institute has at least 167 sequencing machines [31] and projects with names like the "Million Human Genomes Project" [6]. World sequencing capacity was estimated at 13 quadrillion base pairs per day at the end of 2011 [31], around 30,000 human genomes, and there is no sign of slowing. In fact, the growth in the amount of sequencing data is currently growing at a rate faster than Moore's law and is projected to continue for some time [8]. Moore's law states that the number of transistors will double approximately every two years [18]. Thus, new innovations in algorithms and designs that utilize the transistors more efficiently are needed if the aligning of the data is to keep pace with the sequencing machines.

Numerous software projects have been developed to align the short reads from the sequencing machines with a reference genome, some of which are discussed in Chapter 2. A comparatively fast aligner called SOAP3 was release in 2012 from the University of Hong Kong. To handle last year's world capacity, it would take at least 300,000 GPUs running SOAP3 constantly, let alone process the back log. Because of the large amount of sequencing data, it also becomes important for the processing of the data to occur on site. Networks have already become a bottleneck to offsite computer clusters, resulting in some scientists sending their data

on hard drives via FedEx to be processed [31].

RAMPS (Reconfigurable Architecture for Minimal Perfect Sequencing) offers a new approach utilizing the Convey HC-2, a hybrid core computing system. Using the human genome as a reference, RAMPS aligns short reads on the order of hundreds of thousands of times faster than software such as SOAP2 or Bowtie, and about 1000 times faster than GPU implementations such as SOAP3. By decreasing the preprocessing time, we hope to fundamentally change the computational problem by allowing a higher percentage of the read data to be aligned exactly to the reference. This is made possible by the use of a highly pipelined hardware design and the large amounts of memory bandwidth provided by Convey's hybrid core computing system.

This work provides discusses related work and projects, presents relevant background material, details the implementation of RAMPS, and reviews and discusses the results in comparison to other recent works. Chapter 2 is a survey of current software and hardware short read aligners and a discussion of the algorithms currently being used in the field. Chapter 3 contains background knowledge of the minimal perfect hashing algorithm and DNA sequencing in general; two important topics for understanding the work. RAMPS contains two main components implemented both in software and hardware: a series of hardware pipelines to preprocess a reference genome into a hash table, and a hardware aligner for performing fast lookups. Both of these designs and their implementation are discussed in detail in Chapter 4.

CHAPTER 2. REVIEW OF LITERATURE

There a plethora of short read aligners being published in research field of bioinformatics. They vary greatly in the length of reads they sequence, data formats allowed, and algorithms used for matching. Many aligners attempt to have high sensitivity, or the ability to find matches given mutations in the DNA. Often, the user is allowed to decrease the sensitivity of the program in order to improve the execution runtime. Table 2.1 lists some of the more often sited short read aligners from the literature, including self-reported timing, memory usage, and algorithm selection.

Table 2.1: Comparison of popular short read aligners.

Tool	Platform	Speed (reads/s)	Algorithm(s)
BLAST [3]	CPU	-	Database, Smith-Waterman
MAQ [26]	CPU	50	Hashing
SOAP2 [27]	CPU	2,000	FM Index
Bowtie [22]	CPU	2,500	FM Index
BWA [24]	CPU	10,000	FM Index
SOAP3-dp [21]	GPU	200,000	FM Index
BWA-Convey	Convey	350,000	FM Index
BWT-GPU [32]	GPU	400,000	FM Index
BFAST [16]	CPU Cluster	700,000	Multiple Indexes, Smith-Waterman
RAMPS	CPU	800,000	MPH
RAMPS	Convey	315,000,000	MPH

Early solutions for DNA sequence alignment were slow. Some used a database of common sequences or a hash table to help align new data. BLAST [3], or Basic Local Alignment Search Tool, was published in 1990 and was one of the first tools available for sequence alignment. MAQ [26] was an implementation that provided quality data along with its alignment results. Unfortunately, many of the early tools became obsolete because of their slow alignment speed when the new generation of sequencing machines became more popular.

A newer approach, used in SOAP2 [27], BWA [25], and Bowtie [22], is to index a reference genome using an FM Index [12] which compresses the genome using the Bowler-Wheeler transform [9]. This scheme allows the genome to be compressed in a suffix tree, reducing the memory footprint, and allowing use of commodity hardware. Unlike a hash table approach, indexing in this way creates an algorithm where getting the alignment data is the result of a pointer-based tree transversal. In cases of a mismatch, time-consuming backtracking is used to find segments that may match with high probability. Applications like SOAP2, BWA, and Bowtie are generally similar in their approach, but differ slightly in the way that they construct their index of the reference genome and optimize the algorithm. They are all, however, limited by memory bandwidth because of both the tree traversal and the big data nature of the problem.

Our approach differs from prior implementations in a number of important ways. Prior approaches seem focused on using commodity hardware, reducing memory footprints, and providing algorithms for finding the matches for reads that contained mismatches or fuzzy data. RAMPS takes the approach of reducing memory bandwidth to the extreme. By preprocessing the reference genome into a hash table, alignment of arbitrary reads is a simple hash table lookup. This has the advantage that each read aligned has a small cost in terms of memory usage: 4 load operations to retrieve a 100 base pair read, 2 loads from the hash table, 4-5 loads from the reference genome, and 1 store operation to a results array. For aligners based on the FM index, the number of memory operations is usually proportional to the length of the read for exact matches. The aligners can search multiple paths of the tree in order to allow for mismatches and indels, but this process results in considerably more memory operations and slower execution time.

There are additional ways to speed up BWA type aligners that would result in less of a drop in sensitivity, such as combining nodes on different levels of the tree, or by using a hash table on the starting segment of the tree transversal. Such an approach, using a hash table for the first few levels of the tree, was demonstrated by Arbabi et al. [4] at the MEMOCODE 2012 design competition and resulted in reasonable speed improvements without greatly compromising sensitivity.

Most implementations using alternative hardware follow the mainstream approaches and simply tweek and port the existing algorithms to work on GPUs, CPU cluster, and FPGAs. For example, a group from Virginia Tech ported the slower RMAP algorithm to the GPU [2]. Even though the RMAP algorithm was a good fit for the GPU architecture and they improved the performance tenfold, the newer algorithms running on commodity were faster. Torres et al. took the BWA source and ported it to a GPU [32]. Performing only exact matches, they were able to gain similar performance increases as SOAP3. SOAP3 [28] uses the same approach, but allows for higher sensitivity. Both utilize the GPUs higher memory bandwidth for greater speed, but they fail to fundamentally change the algorithms used.

Because prior attempt only changed the hardware used, not the algorithm, the result is a marginal increase in performance due to the use of better hardware. To achieve big performance gains, more must be done than just tweaking the same algorithms to work on better hardware. In fact, since the growth of DNA sequencing data is currently outpacing the growth of Moore's law, simply using better hardware won't solve the problem in the near future. RAMPS breaks the tradition of the previous short read aligners. The algorithm RAMPS uses for alignment is very simple and scales remarkably well due to the redundant nature of performing a single instruction on multiple pieces of data.

CHAPTER 3. PRELIMINARIES

3.1 Background on DNA Sequencing

DNA (Deoxyribonucleic acid) is double helix composed of four nitrogen based nucleobases: Adenine, Thymine, Guanine, and Cytosine (abbreviated ATGC). The DNA molecule actually contains two copies of the genetic information; an important attribute that allows it to be easily replicated by splitting the two chains of the double helix apart. The two chains run anti-parallel to each other, with one end of a single chain labeled 3' (three prime) and the other labeled 5' (five prime), depending on the direction of the 3rd and 5th carbon atom on the sugar molecule. The bases (ATGC) of each chain pair with one another using hydrogen bonds. Adenine always pairs with Thymine (AT); Cytosine always pairs with Guanine (CG). While the relative proportion of the bases in DNA were known to be approximately equal in the base pair groups, Watson and Crick were the first to proposed the double helix architecture in which the bases actually bonded in 1953 [33]. Due to the antiparallel nature of the chains and base pairing, given one chain of DNA you can compose the opposite chain by reversing the order and substituting base pairs. The base pairs are chained together using a 5 carbon sugar (ribose or 2-deoxyribose) called a nucleoside; the nucleoside and nucleobase are together referred to as a nucleotide. In the human genome, there are over 3 billion base pairs. They are grouped into 23 chromosomes, each containing hundreds of millions of base pairs. Full genome sequencing is the process of trying to discover the exact sequence of base pairs for a particular individual.

In general, sequencing DNA today involves breaking the DNA into small segments, amplifying, splitting the chains apart, and rebuilding one of the chains one base pair at a time. Amplification can take a single segment and multiply it millions of times using a polymerase

chain reaction (PCR) machine, based on a process of repeated heating, cooling, and duplication accredited to Kary B. Mullis [29]. The amplification is necessary so that when fluorescently marked bases are added to a sample when rebuilding one of the chains a single nucleotide at a time, a laser can detect which base pair was added. Newer models of sequencing machines perform "paired end" reads; i.e. both chains of the double helix are sequenced in order to improve quality.

Next generation sequencing devices, such as the ones from Illumina [17], can produce a massive amount of data: 300 million to 3 billion short paired reads on a typical 1 to 11 day run of the sequencer. Typically, the base pairs in the genome are sequenced multiple times to ensure that every portion of the genome is sequenced, since it is hard to know exactly how the DNA is broken into small segments. The amount of duplicate data, called coverage, is typically in the range of 2x (low coverage) to 20x (deep coverage) [11]. Thus, it is normal to have tens of billions of base pairs worth of information to align.

To align the short sequencing data, a reference genome is used. The reference is like having a picture of the puzzle while attempting to put the puzzle together. Aligners take each short read from the sequencing machine and attempt to ascertain its position in the reference genome. At best, this amounts to a simple string matching. At worst, it involves allowing for mutations in the read data such as mutations (single nucleotides with a different base), indels (insertions and deletions of single nucleotides), or gapped alignment (allowing for large gaps in either the read or the reference).

3.2 File Formats

Sequencing data typically comes in a text (ASCII encoded) data format known as FASTQ. The FASTQ format is discussed in detail in Figure 3.1. RAMPS utilizes programs from the MEMOCODE 2012 reference design to compress the reference genome into a binary encoded form. A single ASCII encoded character (such as an 'A') typically consumes a single byte of data (8 bits). Since there are only four bases, each base can be binary encoded into 2 bits; thus, the sequence data can be compressed by at least a factor of 4. The data is further compressed by removing axillary comments and quality data removed. Since we wrote our own hardware using the Convey system, we are better able to take advantage of this compression, as the hardware can manipulate 2 bits at a time whereas a normal compute must manipulate data by the byte (8 bits) or word (32 bits).

The quality score is also ASCII encoded. In ASCII, each character is assigned an 8 bit binary value. The value of 'E' in binary is 0b01000101, or 69 in decimal. 'F' is 0b01000110, or 70 in decimal. Sequencing machines map the probability that a given base in incorrect into a set range of numbers, and then store them in the FASTQ file appropriately. The higher the number of a quality character means the higher the quality. Thus, 'F' is of higher quality than 'E'.

```
@ERR050082.521 HS18_6628:6:2303:13171:165808#3/2
ATTCTCCTCCAAGGCTGCAGAGGGGGCAGGAATTGGGGGTGACAGGAGAGCTGTAAGGTCTCCAGTGGGTCATTCTG
+
:DDFGFCFHEELJMJHIFDEOHHFIKDFIK;CEILH@NIAK:LHIKMIJ9HILBJJGJII7HIHJFJJGCJFGI?CI
```

Figure 3.1: The FASTQ file format stores short read data and quality. Each short read occupies four lines of text, two of which contain actual data. In this example [11], the first line starts with an '@' and allows an optional comment with details about the sequencing machine, a unique identifier, and information about this particular run. The next line contains the sequence data encoding of the nucleotides in the familiar ACGT abbreviation. For this machine, the reads are 100 base pairs in length. A third line delimits the bases from the quality data with a plus sign. Finally, the quality data is given, which basically provides the probability that a given base pair is wrong.

There also exists a format called FASTA in addition to the FASTQ format for sequence

data. FASTA is used for storing aligned reference data, such as the human genome. The FASTA format similarly uses ASCII encoding for the reference sequence. In addition to the regular nucleic acids (ACGT), there were another few extra characters in the reference we used (human g1k v37 [1]). The most common, 'N', represents unknown base pairs. In the reference human genome we used, roughly 8% of the genome was unknown. Lines beginning with a '¿' character are used to mark the beginning of a sequence and can contain useful comments (such as labeling chromosome 1 and indicating it has 249,250,621 base pairs).

```
>1 dna:chromosome chromosome:GRCh37:1:1:249250621:1
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
...
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNTAACCCTAACCCTAACCCTA
ACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTA
...
AACCAGGTCCAGGAAGAAGGTGCAAAGACAGCATTCCAGGTAAAAGAAACAGCTTGAACA
AAAAGTGTGTAGGGGAACCGCAAGCGGTCTTGAGTGCTGAGGGTACAATCATCCTTGGGG
AAGTACTAGAAGAAAGAATGATAAACAGAGGCCAGTTTGTTAAAAACACTCAAAATTAAA
...
```

Figure 3.2: The FASTA file format stores reference data, such as the human genome. The lines are 60 characters long, with each character representing a single base pair. 'N' means that the base pair is unknown [11]. The ellipses are for clarity and not actually part of the file format.

3.3 Minimal Perfect Hash Algorithms

A hash table is a type of associative array for storing data based on the hashed value of a key. A hash function is a series of operations performed on the key that maps that key to a specific index in the table. Minimal perfect hashing is a way of building a hash table for a fixed set of keys without collisions (perfect) and without wasted space (minimal). Collisions occur when two different keys hash to the same index. More formally, a minimal perfect hash is a hash function h from a set of keys S to a range of numbers range $[n] = 0, ..., n1$ where h is $1 - 1$ on S. For a small amount of keys, finding such a hash function is possible by randomly searching and checking. For larger sets of keys, generalized algorithms use an intermediate table of values to adjust the hash function subtly in order to eliminate collisions and create the minimal perfect mapping. Figure 3.3 illustrates an example of a minimal perfect hash on set of known keys.

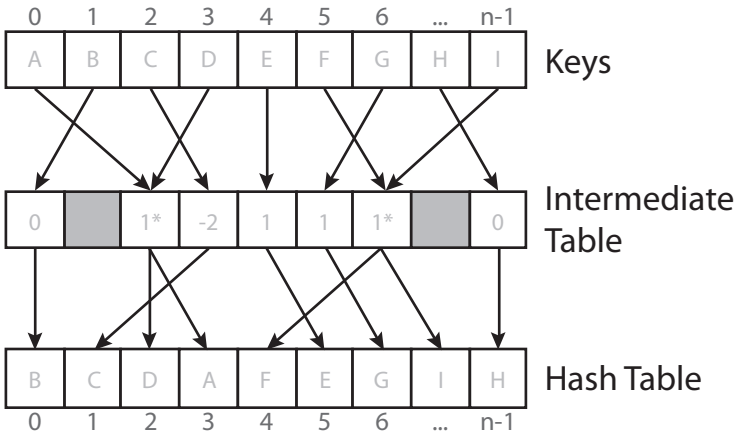

Figure 3.3: An example of a Minimal Perfect Hash (MPH) table. Each key is initially hashed once to retrieve a few bits of information stored in an intermediate table. The key is then either rehashed with the new seed from the intermediate table, or the offset from the intermediate table is added to the initial index. The MPH is created by choosing values in the intermediate table so that a given set of keys will not collide, i.e. two keys will not be directed to the same index.

For any given key, the probability of it hashing into any given index should be equal. Hash functions are designed so that they provide the same index given the same key. However, many hash functions allow the use of a seed value. Changing the seed creates an entirely new mapping

from the key set to a set of indices. This becomes important when a collision occurs in the minimal perfect hashing scheme, as it allows the intermediate table to choose a new seed that will cause the colliding indices to not collide.

Linear time algorithms for the construction of general minimal perfect hashes became practical starting in the 1990's, but their construction was complex. Botelho's dissertation provides a history of the algorithms [7], starting. Schmidt and Siegel were the first to propose a linear time construction algorithm, though in practice it was impractical. The current dominant algorithm is Compress, Hash, Displace (CHD) [5], though there were other more complex algorithms prior. The majority of recent papers improve the main algorithm's storage complexity by using various compression schemes [13, 14, 35].

The basic approach used in the hash and displace algorithm of creating a minimal perfect hash is outlined in the Algorithm 1 below. Essentially, one chooses values for the intermediate table starting with those locations that had the most key collisions. After assigning new seed values to locations that had collisions, the remaining keys are displaced into open locations in the table.

Algorithm 1 Pseudocode of MPH Creation

```
1: procedure CREATEMPH(keys, table)
2:     Count the number of keys that fall into each slot of the table
3:     Sort the keys into buckets in falling order of the count from the previous step
4:     for (each bucket in order of size (where size > 1)) do
5:         repeatbucket.seed++
6:         until (All keys in the buckek now fall into empty spots in the hash table)
7:         Record the new seed value
8:         Mark the location of the keys (using the new seed) as occupied
9:     end for
10:    for (each bucket in order of original location in table (where bucket size == 1)) do
11:        Let i be the location of the next available index in the hash table
12:        Let j be the location of the single key in the hash table
13:        Record the new offset value (i − j)
14:    end for
15: end procedure
```

CHAPTER 4. IMPLEMENTATION

This chapter discusses the various algorithms and hardware designs used to create RAMPS. The design is discussed in great detail. It is important to remember that RAMPS is composed of two major components: the creation of a minimal perfect hash table and a simple aligner that uses the hash table to perform look ups. These two components were implemented as both a software solution and a hardware solution.

4.1 Approach

When surveying the plethora of available aligners, it becomes clear that memory bandwidth is main constraint on speed. Thus, RAMPS is optimized for reducing memory bandwidth. The choice of a hash table as the main data structure and of using the Convey's Hybrid-Core computer were based on the need for increased memory bandwidth. Hash tables minimize overhead, requiring only a few memory operations per read to find an index in the genome. The Convey HC-2 was chosen because of its large available memory bandwidth of 80 GB/sec.

The main issue with regular hash tables is the size of the table when scaled to bioinformatics projects. The table size is proportional to the size of the reference genome since there would be one entry per unique 100 base pair sequence in the reference. We used a reference genome from the 1000 genomes project [1] which contained about 2.8 billion unique 100 base pair segments. This means that every byte of data per entry increases the size of the hash table by 2.8 GB.

If we were to store the key (the short read) in the hash table, we would need about 32 bytes per entry, or a table size of over 80 GB. In addition, any collisions would necessitate a linked list type of structure for collision handling, resulting in more increases in the size of the table. There is no need to store the key (actual read data) in the hash table. We store the read's

index to the reference genome in the hash table and use the reference genome itself to make sure the read belonged in the hash table. Additionally, minimal perfect hash functions are only for a fixed set of keys. Short read alignment uses a static reference. Thus, we eliminated the need for collision detection by using a minimal perfect hash function for the table. Figure 3.3 and Algorithm 3 discuss the use of minimal perfect hash tables and their use in performing short read alignments.

The software was written first along with a testing framework. The MPH creation algorithm closely follows that of Belazzougui et al. [5], except we do not compress the intermediate values. A suitable hash function was choosen, Jenkin's Spooky hash [19], that would map easily into hardware. A pictoral explanation of generalized minimal perfect hash construction can be found at Hanov's website [15]. The general process for creating the hardware was a multi-step process: pick good algorithms, draw the designs, create a software model for the Convey software simulator, create the components using Xilinx's Core Generator or Verilog, test the user made components for correctness and timing with a test bench, combine the components together and test using Convey's hardware simulator, synthesize and test the bit files. The revised design drawings of the hardware are presented later in this chapter.

4.2 Overview of Convey Architecture

RAMPS's hardware pipeline was built for use with the Convey hybrid-core computing platform containing a minimum of 32GB of coprocessor memory. The Convey HC-1 and HC-2 are hybrid computers, containing a regular motherboard and a coprocessor board that contains a set of 14 FPGAs (Figure 4.1). Eight FPGAs are wired as memory controllers (MCs), two are used as an Application Engine Hub (AEH), and the remaining four are programmable and called Application Engines (AEs). The host (x86) processor can send the AEH custom instructions, which will then load a custom bitfile (branded as a "personality") onto the AEs and execute the custom instruction. The coprocessor contains its own memory, though the host processor and coprocessor can share all memory in a cache coherent manner. More information can be found in Convey's documentation.

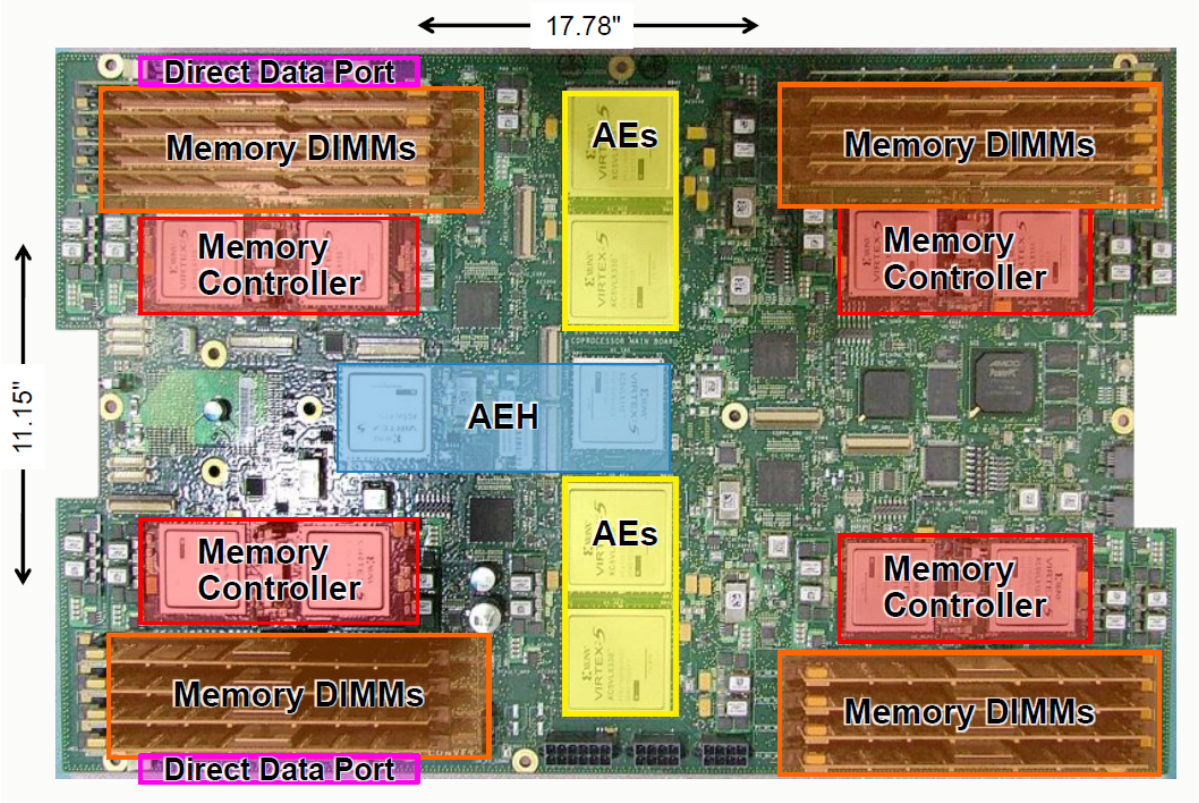

Figure 4.1: The Convey system consists of both the coprocess board (pictured) and a regular commodity server [10].

The distinct competitive advantage that the Convey system provides is its actual 80 GB/s of memory bandwidth. Figure 4.1 shows the 16 memory DIMMs available on the coprocessor board that allow it to access large amounts of memory quickly. In addition, the Convey system has "scatter-gather" DIMMs, allowing random access to memory locations with speed on par with sequential access to memory. In addition to raw hardware speed, Convey provides a rich set of hardware interfaces for accessing memory in its personality development kit (PDK). The PDK is built to allow developers to get their programs up and running quickly, without having to spend time reinventing the memory subsystems. Each AE (Application Engine) is given access to 16 memory controller ports, which are multiplexed to the eight MCs. The AEs operate at a clock frequency of 150 MHz, allowing each memory controller port to make 150 million memory requests per second. Using these memory controllers, along with Convey's provided read order queue and crossbar switch, greatly simplified the hardware design by allowing each

MC port to access any address and by creating an ordered data flow.

The development of a program on Convey starts with the software. First, a software emulator is written. This software emulator allows the developer to test the interface between host machine and the coprocessor's AEH (Application Engine Hub). This step typically involves thinking about what memory should be moved or allocated on the coprocessor, what parameters to pass to the application engines, and how this data is transferred to and from the coprocessor through the AEH. It also allows the developer to create a model that can be used in the future to test hardware code. The next step is to design the hardware and create a custom personality. This step involves diagraming hardware layouts, writing and testing individual modules, testing and fixing the larger top level module, and creating a bitfile by synthesizing the written HDL (hardware description language) code.

4.3 Minimal Perfect Hash Creation

RAMPS's minimal perfect hash (MPH) is created using a fairly simple algorithm discussed in Algorithm 1. Before running the algorithm, the first step is to find the set of unique entries that need to be stored in the hash table. This can be done by hashing each 100 base pair word in the reference genome in a number of rounds, throwing away duplicates and storing collisions for processing in the next round. After we have thrown away all the duplicates and processed the left over collisions, we are left with a set of unique keys. With a set of known keys, we can construct a minimal perfect hash using a generalized method called hash and displace [5].

After collecting a set of unique keys, all the keys are sorted using a two pass counting sort. During the first pass, a count of the size of each bucket of the hash table is taken; if a key collides with another key (they both hash to the same value), the bucket's size is incremented. Thus, after the first pass, the number of collisions for each entry in the hash table is known. During the second pass, keys are placed into a buckets array sorted by the size of the bucket. Since the hash function has an equal probability of choosing any given bucket, the size of a bucket is typically orders small that the length of the hash table. For example, the largest bucket size we encountered was 13 when the algorithm was used with the 2.8 billion entries in the human genome. It is because of the small number of unique sizes that we are able to sort

the keys in linear time with a two pass counting sort.

With the buckets now sorted, the algorithm begins to reseed buckets with a size of two or greater. Reseeding ensures that the hash function will be "perfect" and no longer contain collisions. Starting with the largest buckets, the keys in a bucket are reseeded such that they no longer collide and will occupy empty spots in the hash table. The reseed value is stored in an intermediate table and a bit field or bit array is updated to indicated that the locations in the hash table are now occupied. This process continues with the next largest bucket until all buckets containing two or more keys have been assigned a new seed.

Finally, once all buckets containing two or more keys have been reseeded, buckets containing a signle key are placed into open spaces in the hash table. The offset from their original location is recorded in the intermediate table. This can be done without looking at the keys, using exclusively the information gained from the counting sort (the entries in the hash table with a single entry are marked) and the information gained from reseeding (the bit array containing a record of the empty and occupied slots in the hash table). The intermediate table provides a minimal perfect hash for the given key set, and can now be used to add values to the table to complete the key-value association.

Since the algorithm for MPH creation naturally falls into five stages, the hardware for creating a minimal perfect hash table is broken into five major pipelines. Four of the major pipelines require proper memory ordering, atomic increment operations, or atomic test and set operations. These memory requirements can be solved by using a small cache, indicated by a lock on the memory controller in Figures 4.6-4.9. Using out current design, it would be impractical to run these four major pipelines on all four application engines available on the Convey coprocessor board due to the complexity of maintaining cache coherence across all four chips. However, Botelho's dissertation [7] outlines ways of dividing the problem appropriately for a distributive version of the algorithm that scales nearly linearly with the number of nodes added.

The fifth major pipeline does not require special memory requirements, and thus it can be run on all application engines. All five pipelines for MPH construction occupy two personalities: the first personality contains pipelines 1-4 and the second personality contains pipeline 5. It

should be noted that in Figures 4.6-4.12, the data flows from the left to the right of the diagram over the course of time and stages are marked on memory boundaries. Figure 4.2 however, shows the data flow between application engines.

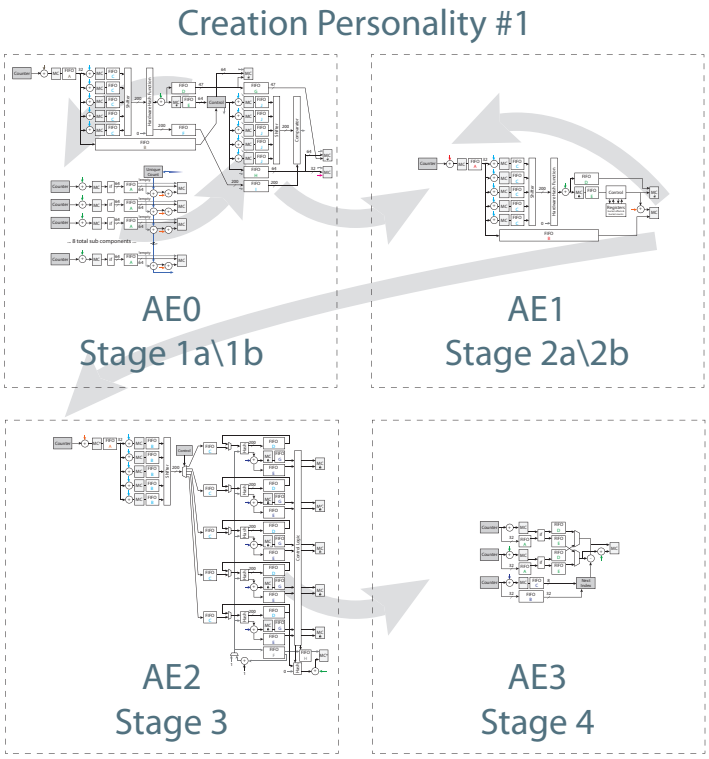

Figure 4.2: RAMPS contains two personalities for creating the minimal perfect hash table. The first features four different bitfiles (total of 6 hardware pipelines). The data is processed by each hardware pipeline sequentially, starting with the first. Each major pipeline must process the data completely before moving on to the next phase. Here, AE0 loops until all unique reads in the genome have been found. Next, AE1 performs a two pass counting sort. AE2 then reseeds large buckets, and AE3 finally displaces the singular buckets.

In RAMPS, the intermediate table and hash table are interleaved. With smaller table sizes (less than 20 million), it makes sense to have a separate intermediate table with compressed values so that the intermediate table can fit into the cache on a processor. In our case, with a human sized genome near 3 billion entries, the extra complexity of compressing the intermediate table was skipped and it was instead placed along with the values of the hash table. In fact, this interleaving may improve performance marginally; offsets in the intermediate table are small and the result may be within a few entries in the table, thus if cache lines on the chip are larger

than 8 bytes it may result in a cache hit or the memory controller can combine operations since the data may reside in the same row of a RAM module. The layout of a single entry can be seen in Figure 4.3:

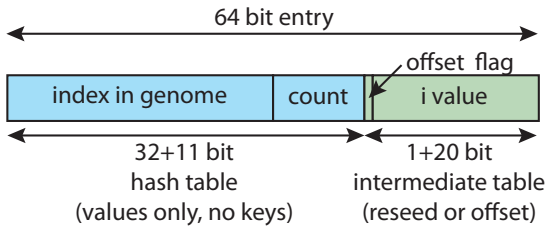

Figure 4.3: An entry in the Minimal Perfect Hash table. The table is an array with one 8 byte entry per unique read in the reference genome. The intermediate table is stored in 21 bits and is used to find a unique index into the table for each key. The first 43 bits contain the value associated with the key; i.e. the index of occurrence in the reference genome and a count of how many times it occurs.

The software package of RAMPS, unlike the current version of the hardware, is configurable using a command line argument to allow creating hash tables of any number of base pairs that is a multiple of 4. The multiple of 4 was chosen because the bit packed representation stuffs four base pairs per byte, thus keeping the data byte aligned.

The default number of base pairs used for creating the hash table is 100. The following commands create a hash table using either the CPU (./create) or Convey coprocessor (./runcp):

```
./create -g human_g1k_v37.bin -h output_table.bin
./runcp  -g human_g1k_v37.bin -h output_table.bin
```

The command line argument 'l' can be used to specify a different size, in bytes. For example, a hash table of the genome using all reads of 36 base pairs can be generated by:

```
./create -g human_g1k_v37.bin -h output_table.bin -l 9
```

Additionally, the '-d' command line argument specifies if the reference genome contains both chains of the DNA and controls whether the hash function produces a normal hash or a paired hash. A paired hash means that both chains of the DNA at a given index will hash to the same value; e.g. hash("AAAG") == hash("CTTT"). The software simply calculates a

read's pair (reverse the sequence and swap base pairs) and hashes whichever input is greater. Paired hashing effectively doubles the number of reads you can align to a reference genome that only contains one of the DNA chains, and is enabled by default.

4.3.1 Hardware Hash Function

Most of the computation of the algorithm occurs in the hash function. Jenkin's Spooky Hash [19] was chosen because it is both fast in software and easy to implement in hardware due to its reliance on only shifts, adds, and XOR operations. RAMPS uses a slimmed down version Jenkins' Spooky Hash that has been stripped to work with only keys of size 25 bytes (100 base pairs), though adjustments could easily be made to handle keys of any given bytes size below a constant maximum. By stripping unnecessary branches and instructions, each hash requires 23 rotations, 23 XORs, 27 additions, and 1 mod operation. By grouping operations together, a 34 stage pipelined hash function was created in hardware.

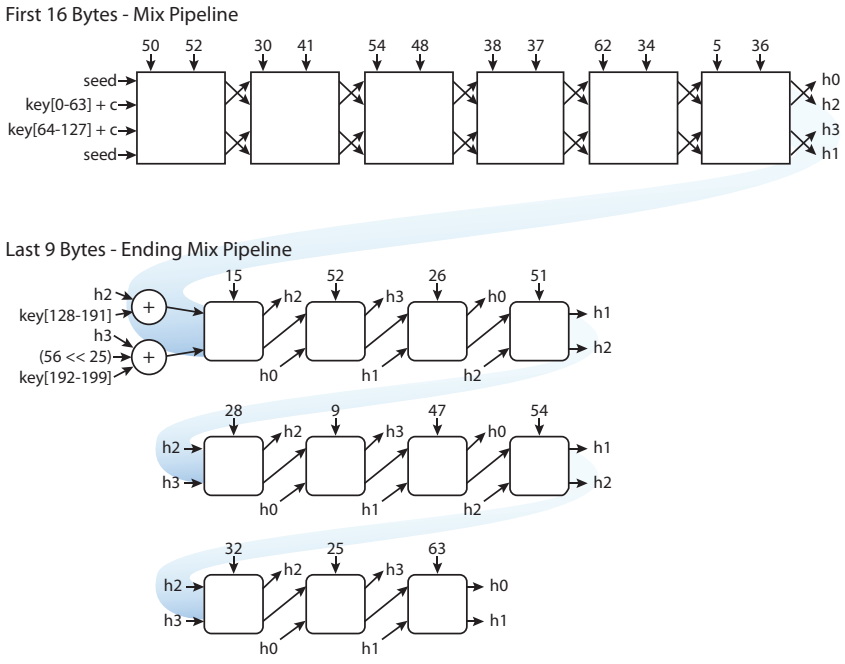

Figure 4.4: The hardware hash pipeline is composed of several blocks which are further described in Figure 4.5. The diagram here shows the mixing of the first 16 bytes followed by the last 9 bytes. The seemingly random constant values help the hash function create a waterfall, where a single bit change in the key being hashed causes all the bits to change with equal probability.

The pipeline stages listed in Figure 4.4 are registered, increasing the maximum allowed clock frequency for the unit. Further registers could be added within the mix blocks for additional speed improvements. The key is fixed at 25 bytes, or 100 base pairs. The hashing algorithm hashes the key 16 bytes at a time, and has specific ending mix blocks for the last 9 bytes.

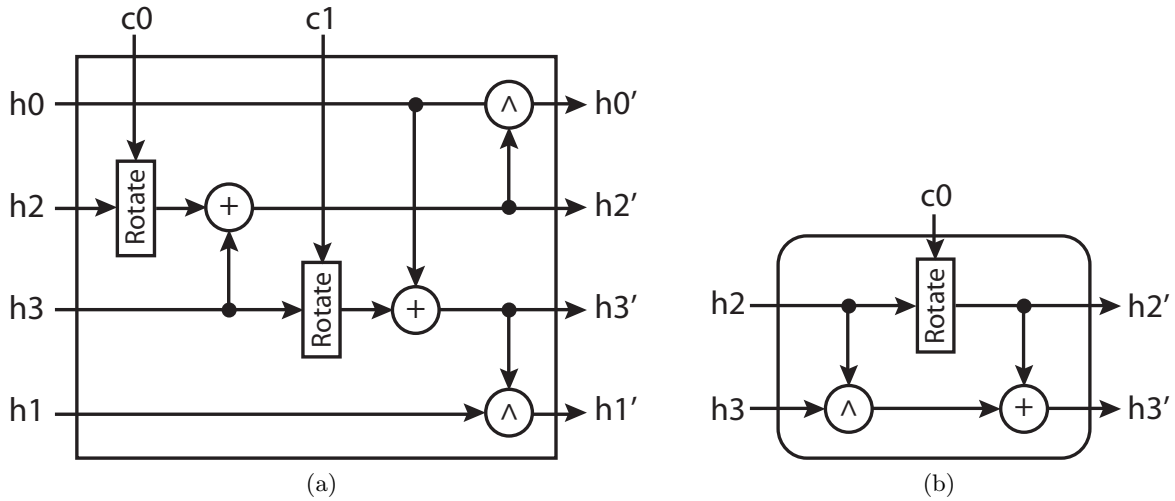

Figure 4.5: The hash function pipeline in Figure 4.4 is composed of several subcomponents shown here. To the left (a) are the mix blocks used for hashing the first 16 bytes of data. On the right (b) is an ending mix block used to hash the last 9 bytes of the key.

Like the software version of the program, the hardware hash function supports paired hashing; i.e. hash("AAAG") is equal to hash("CTTT"). The hardware performs the necessary wire assignments and bitwise not operation as the first stage of the pipeline, and computes the hash of the larger of a read or the read's antiparallel pair. This can be disabled using the command line '-d' flag.

4.3.2 Pipeline 1a/b - Find Unique Entries

The first step is to find all the unique entries that need to be stored in the hash table. This can be done by hashing each 100 base pair word in the reference genome in a number of rounds, throwing away duplicates and storing collisions for processing in the next round. Algorithm 2 shows that the process of removing duplicates.

Algorithm 2 Pseudocode of Pipeline 1a/b

```
1: procedure PIPELINE1A(list, genome, hashtable, collisions)
2:     for (i = 0; i < length(genome) − 99; i++) do
3:         list[i] ← i;
4:     end for
5:     while list not empty do
6:         erase(hashtable);
7:         Pipeline1a(list, genome, hashtable, collisions);
8:         Pipeline1b(hashtable, unique);
9:         swap(list, collisions);
10:    end while
11: end procedure
```

During the first round, all valid indices into the genome are added to a list for processing. During the processing step (pipeline 1a), any portion of the genome that creates a collision in the hash table is set aside in a collisions list for processing in the next round. Due to the random nature of the hash function, roughly one-third of the indices collide and need to be processed in the next round, leading to a total runtime that is proportional to 1.5 times the length of the genome (sum of geometric series).

Figure 4.6 illustrates this process in more detail. At the beginning of each round, the hash table is erased to remove leftover intermediate data from the previous round. First, an index into the genome is loaded from the list. Next, that index is used to load a 100 base pair read from the genome. While during the first round the indices are in order, the random nature of collisions will cause subsequent rounds to be unordered. In stage 3, the read is sent through the hashing pipeline in order to locate the correct index from the hash table. After finding the hash table entry, control flow breaks in one of two directions. Either the entry is empty and can be updated quickly, or there is a possible collision or duplication which must be checked by loading the original index stored in the hash table. Duplication happens when the genome contains multiple copies of the same 100 base pair sequence. Collisions occur the hash of when two different 100 base pair sequences is equal.

An important component to Pipeline 1a, 2a/b, and 3 is the ability to perform atomic read-write operations. This is indicated in the figures using a lock on the specified memory controller ports. The atomic operations are implemented by using an ordinary cache with the addition

of a lock bit. Any read operations to a locked address are placed into a replay buffer and can receive the unlock signal from an incoming write operation with the same address.

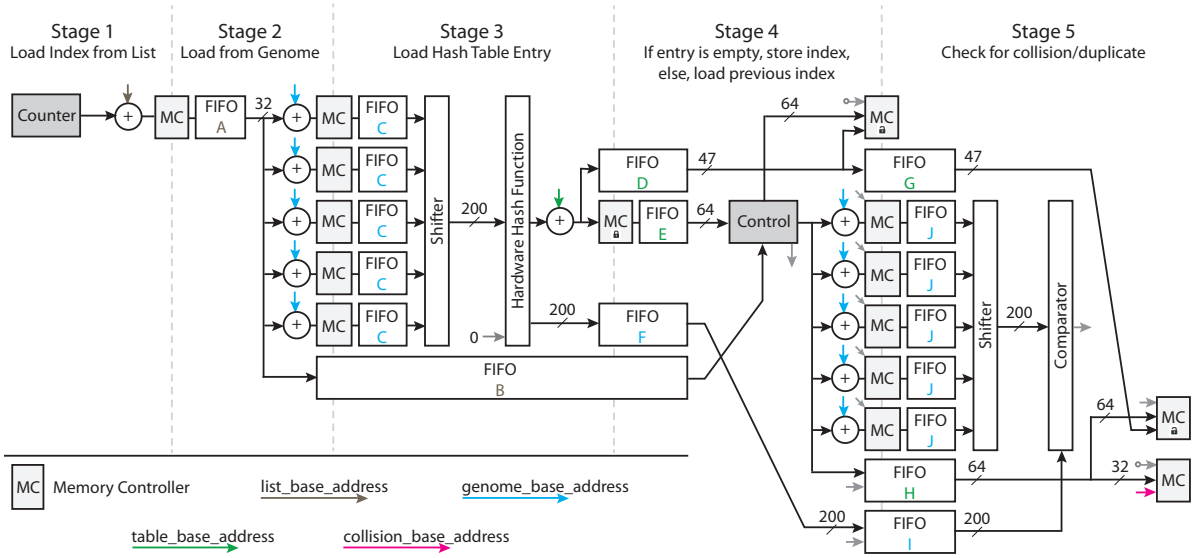

Figure 4.6: This pipeline is broken into 5 minor stages separated by memory operations, shown here flowing from left to right. The pipeline is responsible for processing a list of indices in the genome, removing duplicates and storing collisions for later processing.

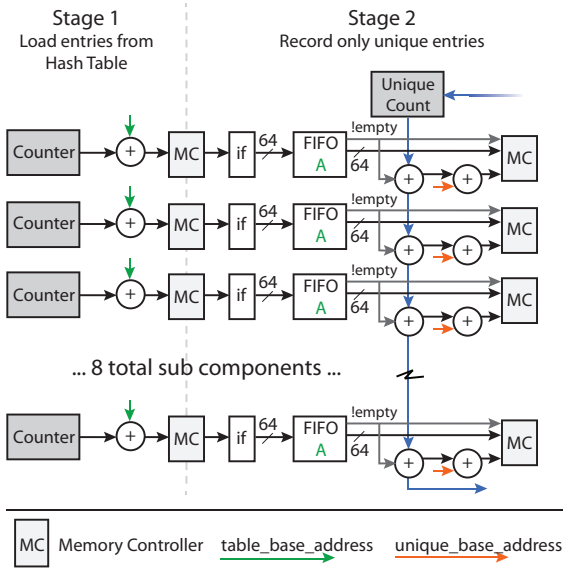

Figure 4.7: The purpose of this piece of hardware is to run through the hash table and collect all unique entries into a list of unique entries. Each of the 8 subcomponents is responsible for loading 1/8th of the hashtable.

After a list has been processed, pipeline 1b (Figure 4.7) quickly runs over the hash table to collect non-empty entries into a tight array. Since the order of the unique reads, or keys, is unimportant, this step can utilize multiple processing units to achieve a significant speedup. This step is necessary; pipelines 2a, 2b, and 5 will each iterate over the list of keys. Removing the empty spaces now reduces the total memory requirement and eliminates memory operations in the future. After copying all the unique keys to a results array, the round ends. If there were any collisions, a new round begins by processing the list of indices added to the collisions list.

4.3.3 Pipeline 2a/b - Counting Sort

With a set of unique keys, construction of the minimal perfect hash (MPH) can begin. As described in Algorithm 1, the first step is to use a counting sort to store the keys into buckets sorted by the size of the bucket. This is accomplished in two passes, but the hardware pipeline is mostly the same for each. The exception is that the 2nd pass stores the unique index into the buckets array, while the first only calculates the size of each bucket.

Before beginning, the hash table is erased. During the first pass, each index is hashed and the corresponding entry in the hash table is incremented by one (indicating the number of indices that hash to a given bucket). In addition, registers on the FPGA hardware are used to keep a running total of the number of buckets of each size. This is accomplished by simply incrementing and decrementing running total counts as each key is hashed. Since the maximum expected bucket size is small [5], we can dedicate a small number of registers for this purpose (RAMPS uses 30). By keeping track of the number of buckets of each size, it allows the algorithm to use a counting sort that runs in time O(n) with respect to the number of keys, n.

After calculating bucket size, the second pass sorts the keys. The initial stages of the pipeline are reused in order to load a unique read, or key, hash it, and load the corresponding value from the hash table. During the last stage of the pipeline, two possibilities exist every time a table entry is loaded: either this is the first key encountered for a given bucket, or it is not the first key. If it is the first key encountered for a given bucket, an appropriate index

is calculated from the running totals calculated in the first pass. This index is then stored in the hash table, and a tally is incremented for the bucket. If there is a tally that is not 0, or an index stored in the hash table, than we know that the key is not the first key encountered for the bucket. In this case, the tally is added to the bucket index stored in the hash table to calculate the appropriate index for storing the unique read, or key, in the sorted buckets list. Figure 4.8 shows the pipeline in more graphic detail.

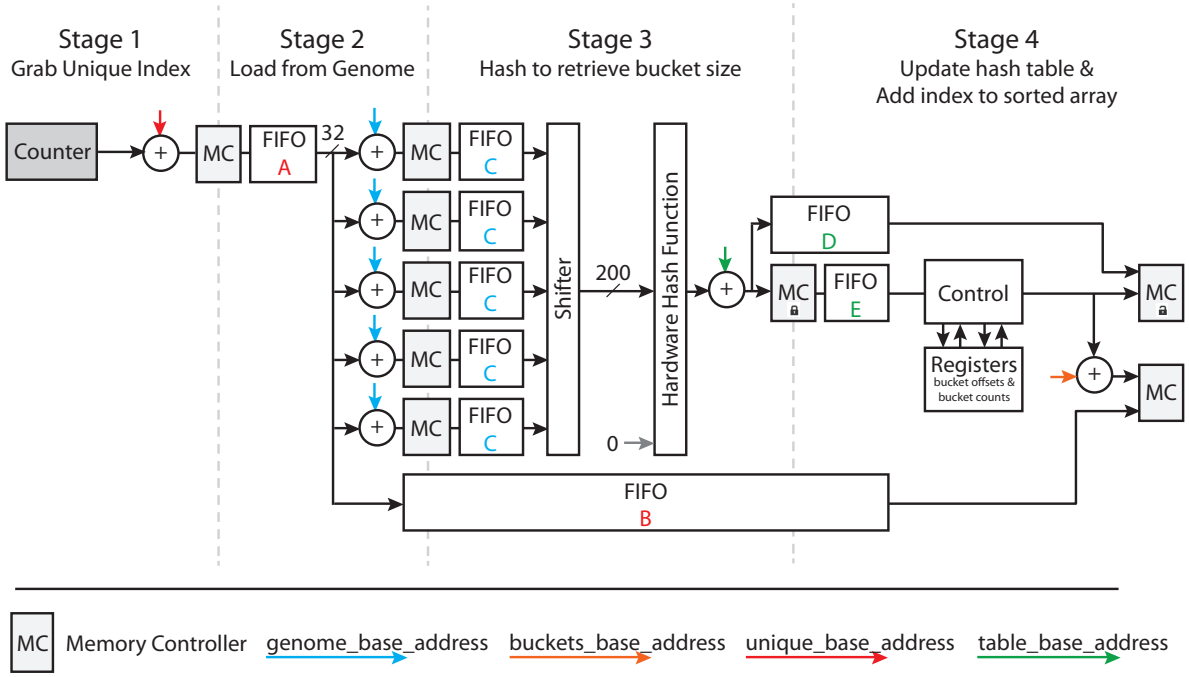

Figure 4.8: For the first pass, FIFO B and the MC port writing to the buckets array are disconnected. The first step is to load a unique reference genome index, which is used to load a 25 byte read from the genome in stage 2. In stage 3, the read is hashed to retrieve an index into the hash table. After the entry from the hash table is loaded, stage 4 does one of two things. If it is the 1st pass, it increments the count and saves the entry. If it is the 2nd pass, the control calculates the correct index into the buckets array to store the index. The hash table is updated with the location of the bucket in the buckets array and the number of indices for that bucket already stored.

4.3.4 Pipeline 3 - Reseed Large Buckets

Starting with the largest buckets, the keys within each bucket are reseeded such that they no longer collide to the same spot in the hash table. The reseed value is stored in an intermediate

section of the hash table. The hardware pipeline was designed to handle buckets containing 2 to 5 keys. There are too few buckets with a size greater than 5 to necessitate designing another hardware pipeline. On a human genome scale, RAMPS software can reseed the buckets with a size greater than 5 in less than 1 second. Buckets containing a single key are displaced in the next hardware pipeline.

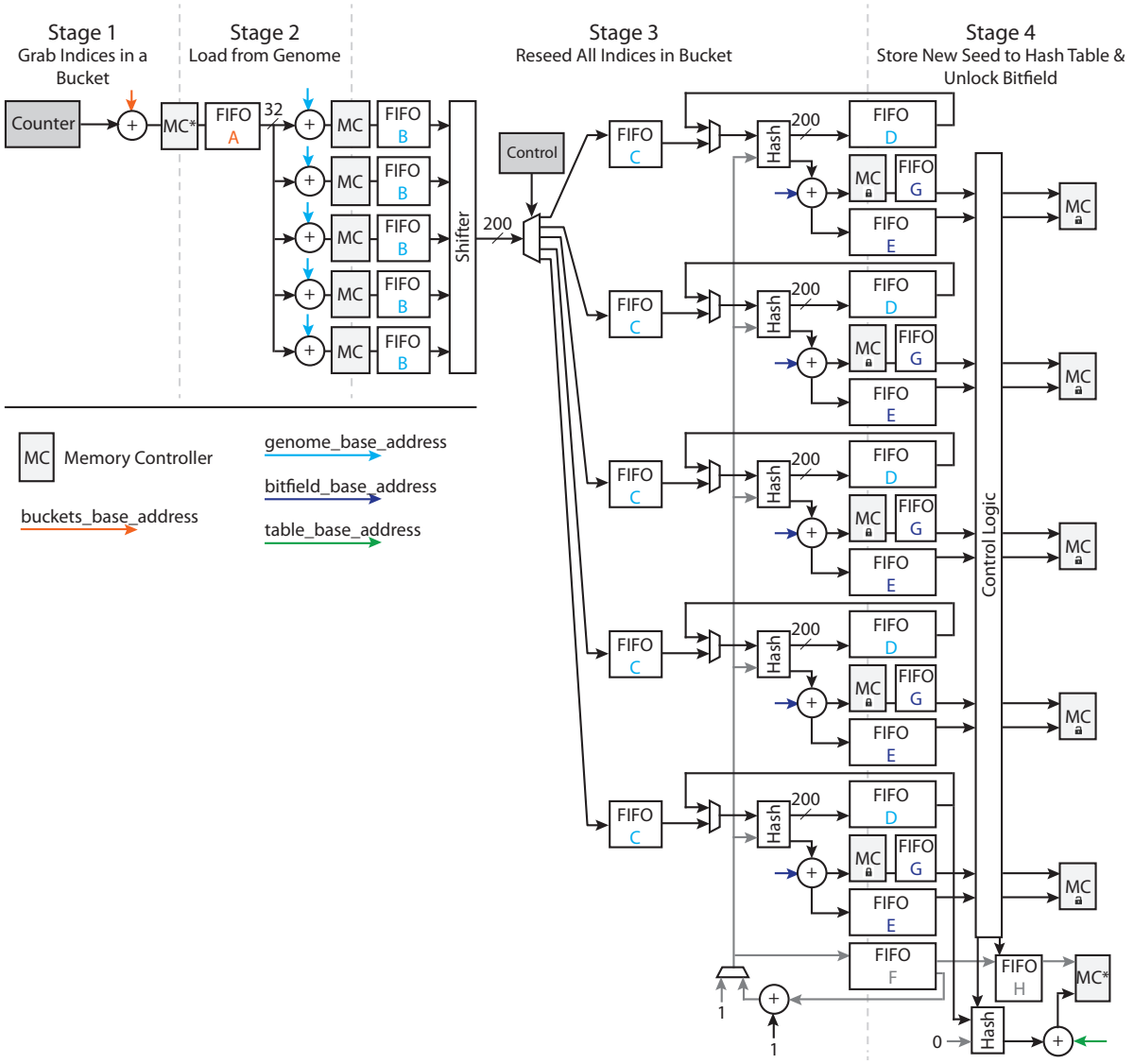

Figure 4.9: The pipeline is run multiple times, once for each size of bucket between 5 and 2. The bucket size controls the multiplexor that stores a read into one of the five FIFOs in bank C. Stage 4 performs a test and set operation on the bit array to ensure that each index is assigned a unique slot in the hash table.

Figure 4.9 the hardware pipeline for reseeding buckets. To reseed a single bucket, all the keys and segments of the genome are first loaded. The reads are rehashed with a new seed for the hash function, which results in the previously colliding keys to have different indices. A bit array is checked to ensure that space is available in the hash table and to reserve the new slots for the keys that were just reseeded. If any of the keys collide when rehashed or if they fall into a taken spot in the hash table, the bucket is reseeded again. To reseed, the seed value to the hash function is simply incremented because a single bit change in the key or seed results in a waterfall change in the hash.

4.3.5 Pipeline 4 - Displace Singular Buckets

Once all large buckets containing two or more keys have been reseeded, the singular buckets containing just one key are placed into open spaces in the hash table. The open spaces in the hash table are those left empty after reassigning all keys that fell into large buckets. The offset from their original location is recorded in the intermediate section of the table.

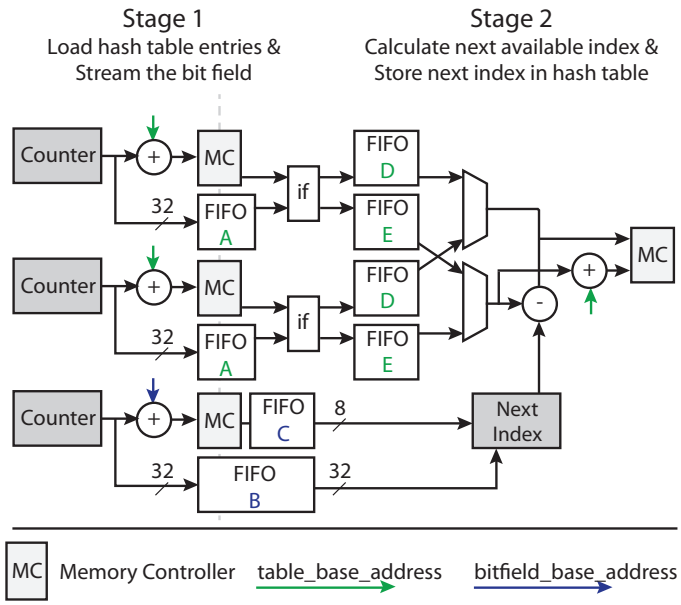

Figure 4.10: The pipeline reads multiple entries from the hash table at a time in stage 1, as not all entries are singular buckets. It also streams the bit array. An indexer calculates the next available index open in the hash table by looking for the next available bit in the bit array. The smallest index from the hash table in the bank of FIFOs D and E is then displaced and the intermediate value is written to memory.

This process, shown in Figure 4.10, must be done in order so that the keys receive small offset values. This is done by running through the hash table and bit array in order. The first entry in the hash table that has only a single key is matched with the first empty index in the bit array. The next single entry is matched with the next empty spot in the hash table, and so on, until the last single entry is matched with the last empty spot in the hash table. After this step is complete, the minimal perfect hash has been created. The only step left is to use the MPH to add the values to the hash table.

4.3.6 Pipeline 5 - Add Values to Table

Finally, this pipeline adds the unique indices and other data to the hash table. Unlike previous pipelines for minimal perfect hash table creation, this pipeline can be run on all four application engines (AEs). RAMPS partitions the list of unique keys in the following manner: AE0 adds keys $0, 4, 8...$, AE1 adds keys $1, 5, 9...$, and so on. The process of storing data in the table is similar to the process discribed in Figure 3.3 and shares similarity with the hardware aligner.

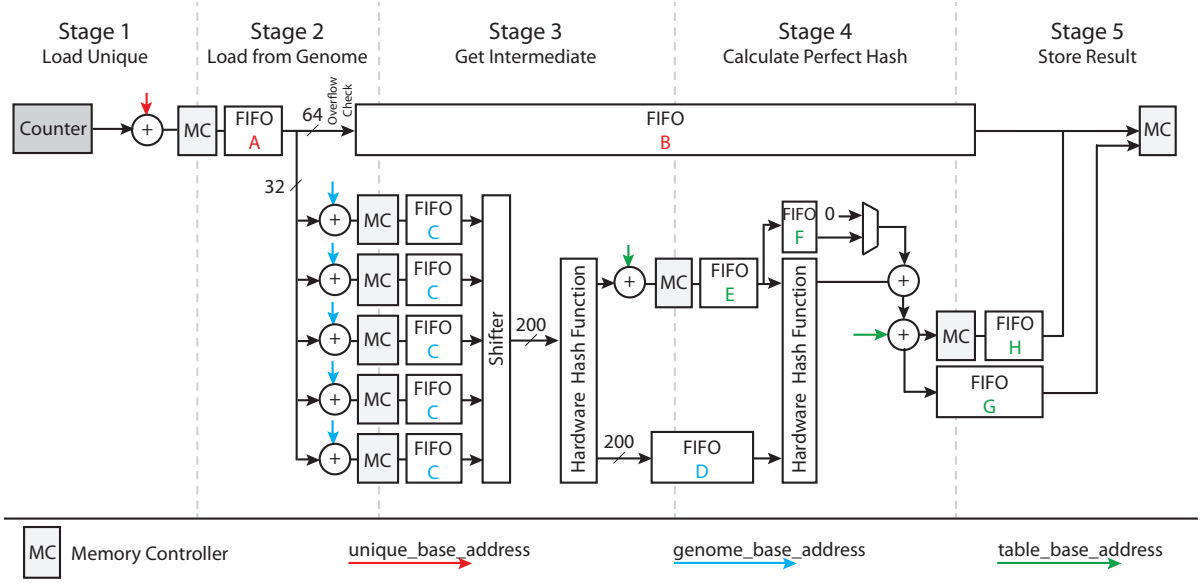

Figure 4.11: The pipeline streams in the entire set of keys (unique indices), loads the genome to retrieve the read, hashes to retrieve the intermediate value, rehashes, and finally stores the data from stage 2 into the table.

4.4 Alignment via Hash Table Lookup

The alignment algorithm is simply to retrieve the correct alignment from the hash table. All possible exact match alignments have been preprocessed from the genome into the hash table. To retrieve an index of occurrence, the program simply attempts to lookup the read in the hash table, validating the index of occurrence is corrected by comparing the read to the genome at the retrieved index. Algorithm 3 formally describes this process.

Algorithm 3 Pseudocode of Hash Table Lookup

```
1:  procedure ALIGN(reads, genome, hashtable, results)
2:      for (i = 0; i < length(reads); i++) do
3:          r ← reads[i];                                   ▷ Stage 1
4:          h ← hash(r, seed = 0);                          ▷ Stage 2
5:          ivalue ← intermediateTable[h];
6:          if (ivalue is an offset) then                   ▷ Stage 3
7:              index ← hash(r, seed = 0) + ivalue;
8:          else
9:              index ← hash(r, seed = ivalue);
10:         end if
11:         entry ← hashtable[index];
12:         check ← genome[entry.index];                    ▷ Stage 4
13:         if (r == check) then                            ▷ Stage 5
14:             results[i] ← entry;
15:         else
16:             results[i] ← NULL;
17:         end if
18:     end for
19: end procedure
```

The five stages of the hardware pipeline for alignment are shown and described in both Figure 4.12 and Algorithm 3. This pipeline is duplicated across all four application engines (AEs). The set of reads is split into four pieces, and each AE contains an identical pipeline to process its subset of the reads.

Figure 4.12 shows the flow of data through the aligner's hardware pipeline. In stage 1 of the pipeline, a bank of four counters and the base address of the read data is used to calculate the address for a given read. These addresses are used to load short reads from four memory controller ports. After a memory latency of about 100 cycles, stage 2 hashes the short reads

to obtain an index for retrieving a value from the intermediate table. In stage 3, the value from intermediate table allows RAMPS to compute an index that is guaranteed not to collide with other entries in the table. The value is either a new seed value for the hash function, or an offset. The unique index is calculated and used to load the value from the hash table. In stage 4, the index in the genome loaded from the hash table is used to load the corresponding 100 base pairs from the reference genome. In stage 5, the reference genome is compared to the read. If they match, the index of occurrence in the genome and number of occurrences are recorded in an output table in memory.

When the pipeline is full, each stage loads or stores data to its memory controller ports on every clock cycle. Much of the combinatorial logic, such as address calculations can occur at the same frequency. The exception is the hardware hash functions, each of which is a 34 stage pipeline. By allowing each portion of the pipeline to perform some part of the alignment on every clock cycle, we efficiently utilize resources and are able to achieve a theoretical throughput of 150 million reads per second per application engine (or 600 million reads per second). In practical tests, the alignment speed was approximately 350 million reads per second.

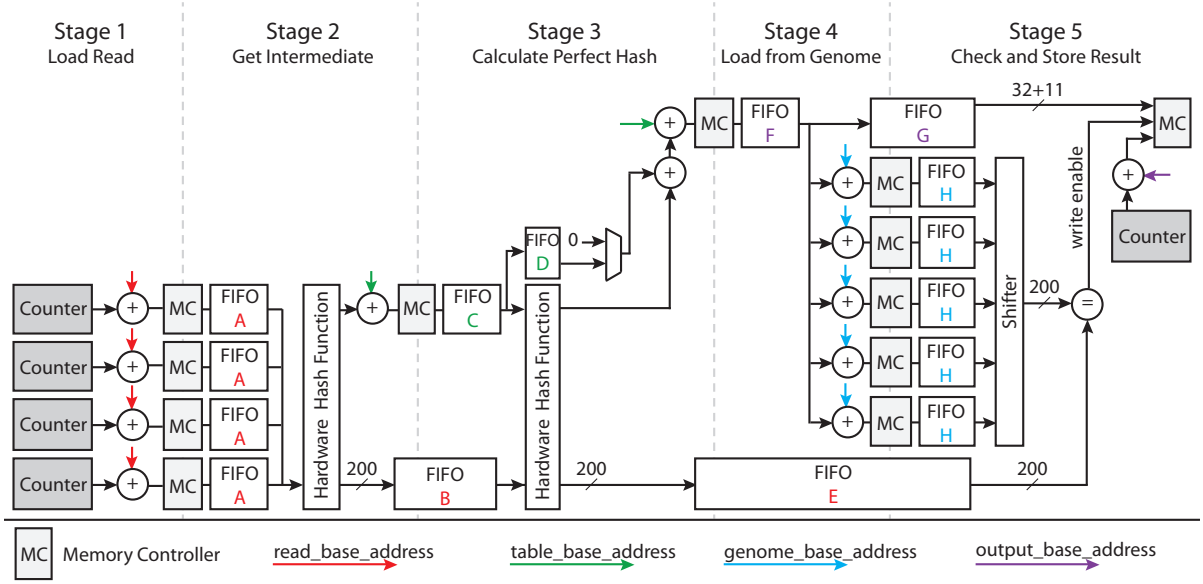

Figure 4.12: RAMPS's hardware hash table lookup pipeline is broken into five stages, shown here flowing left to right.

In addition to the hardware aligner, which can only perform exact matches in its current implementation, the software aligner can be configured to split long reads into smaller parts, each part returning its own index of occurrence. If any indices match, a comparison is done between the entire read and the genome at the matching index allowing for mismatches. This allows the ability of finding matching locations in the reference genome that could be possible locations of the read data. The software aligner takes the following arguments:

```
-l # (read length, default = 25)
-s # (key length, default = read length)
-f # (offset, default = key length)
```

The read length is the length of the read data in bytes. The aligner assumes that the read data is 8-byte aligned. For example, if using the default 25 byte read size, each read will occupy 32 bytes (the last 7 bytes are left empty) in order to align to an 8 byte boundary. The key length is the size of the key in bytes used for creating the hash table. The offset is used by the software aligner to perform more or less hashes in order to improve sensitivity. The following is an example of the terminal commands used to encode fastq data into the binary format and align the data it with three settings for the software aligner: exact matches, a medium sensitivity setting, and a higher sensitivity setting:

```
cat ERR050082.filt.fastq | ./fastq2bin > read_data.bin
./align_exact -g human_g1k_v37.bin -h hashtable_100.bin -r read_data.bin
./align       -g human_g1k_v37.bin -h hashtable_32.bin  -r reads.bin -s 8
./align       -g human_g1k_v37.bin -h hashtable_32.bin  -r reads.bin -s 8 -f 1
```

In the example using the medium sensitivity setting, the read length is the default of 25 bytes, while the hash table was created using 8 byte words from the reference genome (32 base pair). The technique used was to take three 32bp chunks out of the 100bp read, take the index that was in the majority out of the three chunks, and do a comparison between the genome at that index allowing for up to 5 mismatches. Though numbers will vary based upon the quality of read data used, this procedure was able to increase the number of reads aligned by about 20% up to 75%, whereas the number of reads containing exact alignments was 52%.

CHAPTER 5. RESULTS

5.1 Preprocessing Runtime

Most short read aligners perform preprocessing of the reference genome into a more suitable data structure that supports faster queries. Traditionally, it is faster to download the preprocessed data structure that someone else has created that to build your own. Using the index build times listed on various project websites and in papers, we can compare the preprocessing time required before doing alignment to a certain reference genome.

Table 5.1: Preprocessing runtime comparison of popular short read aligners

Algorithm	Platform	Preprocessing	Memory (GB)
Bowtie [22]	CPU	4-5 hours	16
BWA [24]	CPU	3 hours	2.5
RAMPS	CPU	2-3 hours	60
RAMPS	Convey	90 seconds	60

RAMPS's preprocessing time in hardware is orders of magnitudes faster than other approaches. While other hardware hash functions exist and hash tables with billions of entries have been created, RAMPS is comparatively much faster in hardware:

Table 5.2: Runtime comparison of popular minimal perfect hash algorithms

Tool	Platform	Keys/second
CHD [5]	CPU	770,000
BPZ [5]	CPU	910,000
RAMPS	CPU	260,000
Botelho [7]	14 CPU Cluster	4,000,000
RAMPS	Convey	30,000,000

It should be noted that this is not a fair comparison. RAMPS performs the extra steps of

removing duplicates from the genome, indirectly referencing the keys in the genome via index, and processed 2.8 billion keys. Both CHD and BPZ were being tested on a set of 20 million URLs.

5.2 Alignment Runtime

Our short read aligner's runtime is 895 milliseconds for processing 284,881,619 short read sequences from the NA06985 individual using human reference genome g1k v37 [1]. The timer starts before the reads begin streaming to the hardware pipeline, and ends after the last read's index and count are written to the result array. Using alignment speeds from Knodel et al. [20], the University of Hong Kong [21], and various papers, we can compare our runtime with other short read aligners:

Table 5.3: Performance comparison of popular short read aligners

Tool	Platform	Speed (reads/s)	Memory (GB)
MAQ [26]	CPU	50	1.2
SOAP	CPU	70	14.7
SOAP2 [27]	CPU	2,000	5.4
Bowtie [22]	CPU	2,500	2.3
BWA [24]	CPU	10,000	3.5
SOAP3 [21]	GPU	200,000	3.2
BFAST [16]	CPU Cluster	700,000	24.0
BWA-Convey	Convey	350,000	-
BWT-GPU [32]	GPU	400,000	10.0
RAMPS	CPU	800,000	23.3
RAMPS	Convey	315,000,000	23.3

RAMPS's runtime omits the one time costs associated with loading both a 22.5 GB hash table and a 780 MB reference genome in to memory. In addition, the stated runtime does not include the time it takes to load the reads from disk into memory. Convey's internal benchmarking puts the bandwidth from host memory to coprocessor memory at 2.4 GB/second. Since the aligner's pipeline can handle approximately 10 GB/second of read data, the host to coprocesser memory bandwidth is currently RAMPS's bottleneck. With an estimated world sequencing output of 13 quadrillion base pairs per day [31] (40 GB/s in compressed form), a

few dozen RAMPS machines would be able to process all the data.

5.3 Hardware Usage

Another consideration was RAMPS's use of available hardware resources on the Xilinx Virtex 5 LX330 [34]. Each of the four FPGAs on the Convey system has 288 36Kbit block RAMs, 207,360 Slice LUTs, and 207,360 Slice Registers. The data in the table below was collected from the map and place & route reports created after building the various bit files. It should be noted that a significant portion of the FPGA resources are used by Convey's optional memory components, such as the read order queue, crossbar switch, and write complete interface.

Table 5.4: Hardware resource usage per application engine

Personality	Block RAM	LUTs	Flip-Flops	MCs (Rd-Wr)	Write Complete
Simpleton	20%	32%	39%	16-16	Yes
Aligner	30%	43%	45%	11-1	No
MPH Create 1a/b	32%	68%	72%	12-16	Yes
MPH Create 2a/b	28%	67%	69%	7-16	Yes
MPH Create 3	44%	81%	85%	8-16	Yes
MPH Create 4	23%	29%	35%	3-16	No
MPH Create 5	28%	43%	45%	8-1	No

To estimate the amount of resources used by Convey's hardware interfaces to the memory controllers and application engine hub, a superfluous control personality, named Simpleton, was created. The Simpleton personality includes Conveys read order queue, crossbar switch, and write complete interfaces. It increments (reads and writes) a single address from memory on all the memory controller ports and nothing else. Table **??** shows the hardware usage of each pipeline. All designs utilize the crossbar switch and read order queue, but those pipelines that utilize the write complete interface are labeled. Additionally, the number of ports used for read operations and write operations are marked; if one of the operations is tied to zero, some of the logic used for controlling read ordering is optimized out of the design. All memory controller write ports are utilized in the MPH creation pipelines 1-4 because these pipelines implement an erasing function in addition to their more complex pipelines.

RAMPS's alignment runtime, like many of the other short read aligners, is memory bound. The current design uses 12 out of the 16 available memory controller ports. Putting multiple pipelines on a single AE in order to use all available memory bandwidth would increase complexity and lead to only a marginal improvement of performance.

5.4 Alignment Percentage (Sensitivity)

RAMPS's hardware package is currently designed for a constant read length, though there are simple tweaks that could be used to support creating a hash table for read lengths less than 100 base pairs. RAMPS's software package includes options to create hash tables based on any length of read that is a multiple of 4 (so as to align to the byte boundary). When performing alignments, long reads can be split into smaller parts, each part returning its own index of occurrence. If any indices match, a comparison is done between the entire read and the genome allowing for mismatches.

In the following table, the read length is 25 bytes (100 base pairs). The hash table was composed of all 32 base pair words in the human genome, and alignment would divide each read into three parts: bytes 0-7, 8-15, 16-23. By reanalyzing the reads that didn't have exact matches, we can improve the alignment percentage beyond exact matches:

Table 5.5: Alignment comparison of popular short read aligners

Tool	Platform	Max Alignment %
MAQ [26]	CPU	93.2
SOAP	CPU	93.8
SOAP2 [27]	CPU	93.6
Bowtie [22]	CPU	91.7
SOAP3 [21]	GPU	96.8
RAMPS	CPU	75.4
RAMPS	FPGA	52.4

The hardware (FPGA) implementation performs only exact matches, while the software aligner allows up to 5 mismatches, though it is not guaranteed to find the best location or report all locations. By fundamentally changing the preprocessing time, we can encourage the use of reference genomes that more closely match the read data; e.g. by using a parent or

relative's DNA. Combined with an increase in read quality, RAMPS can align more of the data.

CHAPTER 6. CONCLUSION

6.1 Summary of Results

RAMPS offers the bioinformatics community a new tool for fast exact match short read alignment with minimal preprocessing time. RAMPS is both the first known implementation of a generalized minimal perfect hash creation algorithm using FPGAs and an award winning, first place, exact match short read aligner [11].

Currently, the number of mismatches between the reference genome and read sequences occur from both imperfect read quality and genetic differences between individuals. The DNA of two individuals differs by roughly 0.1%, or about one base pair out of a thousand [11]. With minimal preprocessing, we fundamentally change the computational problem. Instead of using a generic human reference genome, people may now be able to use the DNA sequence of a blood family member as a reference in order to increase the percentage of exact matches. Higher quality sequencing machines already produce data sets that contain more than 60% exact matches. Thus, as quality improves and with the ability to use references that contain nearly identical DNA, exact match read aligners like RAMPS will be able to handle a larger share of the data.

6.2 Future Work

The alignment part of the problem disappears from the runtime when using approaches like RAMPS. Future work will be needed to pull reads from disk or the network at speeds near 10 GB/s in order to keep such a hardware pipeline busy. While RAMPS offers the potential of increasing the speed of exact matches by orders of magnitude, work still needs to be done for inexact matching allowing mismatches and indels.

One could also perform a comparative power study between the various aligners, though the results would likely be moot. FPGAs are known to be power efficient; their custom hardware means more energy is spent solving the problem and less energy is wasted. The RAMPS architecture spends such a small fraction of time performing alignments and runs at a clock frequency of 150 MHz, and would likely consume a fraction of the energy compared with other aligners.

Additionally, there are other application domains in which the RAMPS architecture could prove useful with minimal modification. In 2011, Google developed their own hash function, CityHash [30], to improve the speed of their hash table lookups at their data centers. A package similar to RAMPS, but allowing for variable length keys, would have been orders of magnitude faster. Minimal perfect hashing is an ideal answer for transforming URLs, which are normally static, into integers in a fixed range. Instead of using B-trees for databases numbering in the billions, minimal perfect hashing could provide solutions for new frontiers of database applications.

BIBLIOGRAPHY

[1] 1000 Genomes. 1000 genomes project: Reference genome g1k v37. `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference/`. [Online; accessed March-2012].

[2] A.M. Aji, Liqing Zhang, and Wu chun Feng. Gpu-rmap: Accelerating short-read mapping on graphics processors. In *Computational Science and Engineering (CSE), 2010 IEEE 13th International Conference on*, pages 168 –175, dec. 2010.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[4] Aryan Arbabi, Milad Gholami, Mojtaba Varmazyar, and Shervin Daneshpajouh. Fast cpu-based dna exact sequence aligner. In *MEMOCODE*, pages 95–98, 2012.

[5] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 682–693, 2009.

[6] BGI. Beijing genomics institute. `http://www.genomics.cn/en/index`. [Online; accessed November-2012].

[7] Fabiano C. Botelho and Nivio Ziviani. Near-optimal space perfect hashing algorithms, 2008.

[8] Adrienne Burke. Dna sequencing is now improving faster than moore's law! `http://www.forbes.com/sites/techonomy/2012/01/12/dna-sequencing-is-now-improving-faster-than-moores-law/`. [Online; accessed November-2012].

[9] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[10] Convey. Convey reference manual v1.1. `http://www.conveysupport.com/alldocs/ConveyReferenceManual.pdf`. [Online; accessed November-2012].

[11] Stephen A Edwards. MEMOCODE 2012 hardware/software codesign contest: DNA sequence aligner. Mar 2012.

[12] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[13] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with o(1) worst case access time. In *FOCS*, pages 165–169, 1982.

[14] Kimmo Fredriksson and Fedor Nikitin. Simple compression code supporting random access and fast string matching. In *Proceedings of the 6th international conference on Experimental algorithms*, WEA'07, pages 203–216, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] Steve Hanov. Throw away the keys: Easy, minimal perfect hashing. `http://stevehanov.ca/blog/index.php?id=119`, March 2011. [Online; accessed March-2012].

[16] Nils Homer, Barry Merriman, and Stanley F. Nelson. Bfast: An alignment tool for large scale genome resequencing. *PLoS ONE*, 4(11):e7767, 11 2009.

[17] illumina, Inc. Sequencing portfolio. `http://www.illumina.com/systems/sequencing.ilmn`. [Online; accessed October-2012].

[18] Intel. Moore's law timeline. `http://download.intel.com/pressroom/kits/events/moores_law_40th/MLTimeline.pdf`. [Online; accessed November-2012].

[19] Bob Jenkins. SpookyHash: a 128-bit noncryptographic hash. `http://burtleburtle.net/bob/hash/spooky.html`, September 2012. [Online; accessed March-2012].

[20] O. Knodel, T.B. Preusser, and R.G. Spallek. Next-generation massively parallel short-read mapping on FPGAs. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 195–201, Sept 2011.

[21] T.W. Lam, Thomas Wong, Yingrui Li, Peking U, and Ruiqiang Li. SOAP3 alignment time. `http://www.cs.hku.hk/2bwt-tools/soap3-dp/`. [Online; accessed November-2012].

[22] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[23] Samuel Levy et al. The diploid genome sequence of an individual human. *PLoS Biol*, 5(10), 2007.

[24] H. Li. Manual reference pages - bwa. `http://bio-bwa.sourceforge.net/bwa.shtml#9`. [Online; accessed November-2012].

[25] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–60, 2009.

[26] Heng Li, Jue Ruan, and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858, 2008.

[27] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[28] Chi-Man Liu, Thomas K. F. Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, Ruiqiang Li, and Tak Wah Lam. Soap3: ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[29] Kary Mullis. Polymerase chain reaction. `http://www.karymullis.com/pcr.shtml`. [Online; accessed November-2012].

[30] Geoff Pike and Jyrki Alakuijala. Introducing cityhash. `http://google-opensource.blogspot.com/2011/04/introducing-cityhash.html`, April 2011. [Online; accessed March-2012].

[31] Andrew Pollack. Dna sequencing caught in deluge of data. *The New York Times*, 2012.

[32] J.S. Torres, I.B. Espert, A.T. Dominguez, V.H. Garcia, I.M. Castello, J.T. Gimenez, and J.D. Blazquez. Using gpus for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 9(4):1245–1256, july-aug. 2012.

[33] J.D. Watson and F.H.C. Crick. Reprint: Molecular structure of nucleic acids. *Annals of Internal Medicine*, 138(7):581–582, 2003.

[34] Xilinx. Virtex 5 family overview. `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`, February 2009. [Online; accessed March-2012].

[35] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3):337–343, 1977.